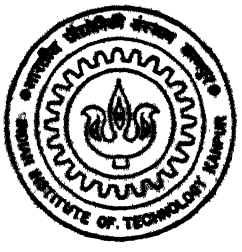
Translating Message Centric OO Specification to C++

by

D. Rafee



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

March, 1996

Translating Message Centric OO Specification to C++

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

D . RAFEE

to the

DEPARTMENT OF

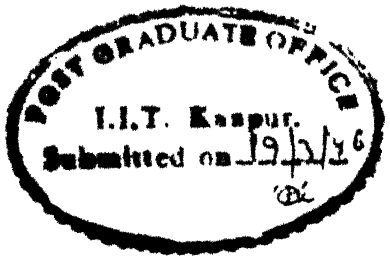
COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

March 1996.

CERTIFICATE

This is to certify that the work contained in the thesis entitled Translating Message Centric OO Specification to C++ by D.Rafee , has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.



Dr. Harish Karnick
Professor
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

March 1996.

Acknowledgments

I would like to thank my advisor, Dr. Harish Karnick, for offering me this thesis. I thank Vijay and Sarma for their invaluable hints and suggestions during this thesis. I am grateful to my parents for being my source of inspiration. I am indebted to my brothers for their constant support in my studies. I thank mech-94 who made my stay at IITK a memorable one.

Abstract

A new method for software specification is described in *A Message Based Specification System For Object Oriented Software Construction* [Sar96]. It represents the system's specification in a formal language. The specification consists of two models, an *Object Model* and a *Message Model*. Classes and relations between classes are given in the Object Model. The dynamic interactions between objects are given in the Message Model. In this thesis we develop a tool that will translate such analysis/design specification into an implementation in C++. This tool will produce C++ headers and code for the methods, where feasible, from the information given in these models. We provide a library for maintaining the persistence of objects in the system.

Contents

Chapter 1

Introduction

1.1 Introduction

Information systems are being developed by using numerous methodologies, techniques and tools. These methodologies and tools have been developed since the beginning of the computer era to support the demand for computer based information systems. To develop a large and complex system successfully, a good methodology and proper tools are essential. A recent trend of viewing systems as collections of identifiable, interacting objects has the potential to initiate a new revolution in information systems development.

Broadly there are two kinds of methodologies namely Structured Design methodologies and Object Oriented methodologies available in the software engineering world. Several OO methodologies are being used for software development these days. Object Oriented Methodologies are still evolving. Some of the well known methodologies in the OO paradigm are by Booch, Rumbaugh's OMT, Jacobson's OO Software Engineering. It is found that software developed by the OO methodology is extendible, reusable and easily maintainable. That is the reason why the OO paradigm is being widely used.

In addition to following a methodology to develop a system, the usage of tools that translate the specification to an implementation enhances the productivity and quality of the software. There exist many such tools for existing well known

methodologies. These tools generate code that satisfies all the design constraints that ensures the reliability of the software produced.

The essential characteristics of object-oriented system development methods are Abstraction, Encapsulation, Modularity and Hierarchy. According to Booch [Boo94] without support for these elements, a model is not object oriented. An object oriented language like C++ provides necessary support for the implementation of these principles.

A new message centric OO method is described in *A Message Based Specification System For Object Oriented Software Construction* [Sar96]. In this method a formal language has been developed for specifying software modules(the analysis/design specification). In this thesis we developed a translator which takes such a specification and generates C++ code for the specified software module.

Classes, Objects and messages/methods are central to the development of object oriented software. A class is a definition of a concept with all of the data items and the operations on that data encapsulated within it. A class generates objects (specific instances) that contain the data and operations for that instance of the class. An operation on data defined within a class is referred to as a method. An object can invoke a method in another object by sending a message to that object.

This method emphasizes both the static structure and dynamic behavior of the system being modelled. Most existing methods place too much emphasis on defining the static structure of the system. The analysis/design specification is represented by a formal textual notation, independent of any programming language.

1.2 Motivation

A Message Based Specification System is a new method, that differs from the existing methods. The aim of this thesis is to develop a tool that will translate the specification produced by this new method into an implementation. The specification produced in this message based method, mainly, contains two parts, an *Object Model* and a *Message Model*. In the Object Model the static structure of the system is specified. Classes, their attributes and relations between these classes are given the

Object Model of the specification. The behavior of the objects is not specified in the Object Model. The behavior of the objects is implicitly specified in the Message Model of the specification. The Message Model gives message passing scenarios between the objects of the system, in terms of *Processes* and *Sub-processes.*

In this thesis, we develop a tool, which automatically generates C++ headers and code for the methods of the objects, where feasible, from the information specified in these Object and Message Models. This tool also generates code for each class to maintain the persistence of objects.

This kind of tool enhances the productivity of the system and quality of the software produced. Manual translation of the analysis/design specification into implementation is an error prone, tedious and time consuming process.

1.3 Overview of the Thesis

This section briefly describes the remaining chapters and appendices. Chapter 2 gives a brief description of concepts of the methodology and syntax of the specification language. Chapter 3 describes rules of translation. Chapter 4 gives the implementation details of the translator. Chapter 5 gives concluding remarks and limitations. Appendix A gives persistence library, a library for support of object persistence. Appendix B gives template library. And appendix C gives the notation of the methodology in BNF.

Chapter 2

Methodology

In this chapter we introduce the concepts of message based specification language and give a detailed description of the notation. The notation is informally specified (The formal specification given in Appendix C). The DOAA Office Automation example will be used to explain various concepts in this chapter.

2.1 Concepts of this Method

In this method the system is viewed as a collection of objects communicating with each other. Each object has some data that constitute its state and some rules that tell when to send a message, when to receive a message and what to do when a message is received . As all these rules involve messages, the rules are specified with the message rather than with the object.

Below, some basic terms that are used in this method are explained.

Object An Object is an abstract or real world entity which has state, behavior and identity, and communicates with other objects.

State The state of an object at any instant consists of its knowledge about itself and its knowledge about the environment at that instant.

The knowledge about itself is represented by the values of the attributes of that object. The knowledge about the environment is represented by the messages it transacts.

message A message forms the unit of communication between two objects.

A message is an asynchronous message as in Actor systems. But not in the same sense as the term is used in many OO Programming Languages (which is essentially a method invocation). There is a difference between method invocation and message. Method invocations are nested (which makes them synchronous), where as messages are not nested (Asynchronous).

The message is defined by the sender, receiver, message identifier and the arguments for the message. Also specified along with the message are Pre conditions (Constraints on the state of the sender to send the message), Post conditions (Constraints on the state of the receiver to receive the message) and Action to be performed by the receiver on receiving that message.

A message can be understood as a command or request one object gives to another object to do some action. Messages are used to represent dynamics of the system in this Method.

Process A Process is a sequence of events that happen to make a meaningful real-world functionality.

A Process can be viewed as a thread of activities. For example, in the DOAA Office automation system, Registration process starts with DOAA sending Registration notification and ends with Student's Roll Number being entered in the Rolls.

Processes and objects are two central concepts of this method. In the first stage (requirements specification) there will be only processes but no classes or objects. In Analysis and Design stages there will be both classes, objects and processes. And in the implementation there will be only objects but no processes.

In requirements specification, processes are represented by plain English sentences. In Analysis and Design stages processes are represented in terms of messages and sub-processes, which again are represented in terms of messages.

Generic process and Generic message Generic process is the mechanism by which we can represent similar turn of events once and reuse it later. Similarly generic messages can be used when similar messages are being sent to different objects.

Generic Processes are used to capture commonalities in the system. For example, most of the queries have same pattern of events : An object A requests for some data in objectB, object B extracts the data and gives the results to object A. Here, except for the participating entities, the sequence of events is same. So, this can be treated as a generic process.

Sub Process Sub process specification is useful to handle complexity. Large processes can be decomposed into sub processes. Generally if there is one central object that interacts with several objects to do an operation then we put these interactions into a sub process.

For example, when Grade reports for all courses reach DOAA, he does "Preparation of Students' grade sheets" sub process. This sub process involves processing the Course grade lists to prepare grade sheet for each Student for the courses he took in that semester, calculating CPI for each Student using CPI Rule from Rule Book, calculating Student's status for the next semester (Regular / Probation / Warning etc.) and finally sending the prepared grade sheets to Departments. DOAA performs this sub process by getting data required from Rule Book and Student.

2.2 Notation

2.2.1 Object model

The object model is used to represent the static structure of the system. There are three kinds of objects in the system : *entity objects, interface objects and control objects.* entity or data objects are used to store information. They usually exist in the problem domain and represent some real-world entity or concept. Interface objects are used to view data in entity objects. Control objects are used to collaborate

between several entity and interface objects. The notation for entity and control objects is same and is different from notation for interface objects.

Class Specification

This specification is valid for *entity* and *control* objects.

CLASS : *Name of the class.*

Name of the class is an identifier[1]. This declaration begins declaration of a new class. All the items that follow this declaration are assumed to belong to this class. Only another "CLASS : xxx" or end-of-file will end the declaration of this class.

TYPE : *ABSTRACT* .

this field is optional. if nothing is specified about the **TYPE**, then it will be taken as non-abstract data class.

An abstract class is one which does not have any instance. A class can be an abstract class only if it is a base class of an inheritance hierarchy.

For example, in the DOAA system, **Person** is an abstract class whose derived classes are STUDENT, FACULTY_MEMBER, DOAA etc. A class derived from an abstract class can itself be an abstract class.

INHERITS : *class name [,class name ...]*

The list of classes from which this class inherits.

ATTR : *attr_name <type> [, attr_name <type> ...]*

list of attributes each specified as *attribute_name <type>*. *type* can be any of

- *STRING, NUMBER, DATE.. etc.* (i.e. scalar type)
- *[attr_decl]* (represents list)
- *TABLE [attributes list]* (a table)
- { *attributes_list* } (set of dissimilar but related items)

[1]Syntactically, an identifier begins with a letter [a-zA-Z_] and contains any alphanumeric character (including "_").

Examples : Person has attribute *name.*

name <STRING>

Grade_Sheet has table of course number and grade.

grades_list < *TABLE [course_no <STRING>, grade <CHARACTER>] >*

GENERALIZATION OF : *class_name [, class_name ...]*
class_name [, class_name ...] are classes which inherit this class.

AGGREGATION OF :

Class name (multiplicity) [, Class name (multiplicity) ...].

RELATED WITH :

RelatedClass name (relation_name, multiplicity)
[, *RelatedClass name (relation_name, multiplicity)*]

relation_name is the role the related object plays with respect to this object in the relation.*multiplicity* specifies how many instances of the RelatedClass are related with one instance of this class.

Interface class specification

The specification of interface classes differs from that of control and entity classes. Details on the specification of interface classes are given in [Sar96].

Only certain fields of interface class specification are considered in the translation of analysis/design specification to C++ headers. For each interface class, name of the class, class type, and related class should be specified.

Eg:

Suppose class COURSE_INTERFACE is an interface class for COURSE class. Specification of the interface class *COURSE_INTERFACE* should be

```
CLASS : COURSE_INTERFACE.
TYPE  : INTERFACE.
RELATED WITH : COURSE(interfacefor,1).
```

2.2.2 Message model

process :: *process_name.*
sub process :: *sub_process_name.*

we divide something into sub processes when there is a central object that is gathering data from several objects to do certain operations.

generic sub process :: *generic_sub_process (parameters).*
INPUTS : parameters_varying_for_generic_sub_process.

The declaration of a set of messages begins with one of the above three **process**, **sub process** or **generic sub process** declarations.

FROM : *Class name.*
TO : *Class name (constraint).*
Class name is the name of the class specified in the object model.
constraint can be

- condition on some attribute of the object. This is used to identify particular instance of the class.

  (variable == class::attribute)
  (class::attribute1 <=> class::attribute2)

- ALL, meaning all the instances of the class should be used for the purpose.

- ALL_RELATED, meaning all the instances receiver class related to this object.

- handle == ref
  handle == BACK

The above means that we explicitly give the handle (*handle* identifies an object in the system. Every object will have an unique handle. In the context of a Programming language like $C++$, handle is same as reference. In the context of a Database, handle is the primary key) instead of fetching it from the relations of the sender object.

When the handle is BACK, this means that the sender should not pick up the object handle from its Relations table, but should just send it, as a response to the message it received, to the corresponding object. For example, any Student can query a Course for its details. Then the Course will extract the details and sends it to Student. While replying, if we specify the Constraint as *name == xxx*, Course will just see whether the Class STUDENT is related to it and checks whether this particular STUDENT exists in its Relations table. If not so, it will not send the message. So, we should specify that even though this Student is not related to the Course, it knows his address through the prior request.

msg_id : *message identifier (parameters).*

The syntax of *message identifier* is same as that of *class name* in **Class specication**. The syntax of the parameters is the same as syntax of "list of attributes" s described in the **Class specifications.**

msg_type : *type of the message.*

type of the message can be reply, shared or exclusive.

reply means the message is being sent as a reply to a message this object has previously received. The sender of the request message expects and waits for this reply message.

shared and *exclusive* represent the security level of the message. If it is specified as *exclusive*, then only this object can send the message to the receiver. If it is specified as *shared*, then other objects can send this message to the receiver only if security level in that message specification is also specified as *shared*. *shared* is useful to restrict the senders to a set of objects.

msg_cond : *constraint that does not depend on the state of the sender, to send the message.*

This is used to send a message repeatedly (looping of the message) or send a message depending on some condition which is not associated with the state of the object.

Example :

- if (variable == value)
- foreach item (table_name or list_name)

result : *reply to this message*

The syntax of "result" is similar to that of list of attributes.

When you specify the "result", it means that the sender "expects" some response to this message. This response is the result of action taken by the receiver of the message.

PRE_COND : *condition on the state of the sender before sending the message*

POST_COND : *condition on the state of the receiver before receiving the message*

It is the combination (and/or) of one or more of the following things.

- received msg_id [of process_id]
- received result_of msg_id [of process_id]
- sent msg_id [of process_id]
- attribute <=> value
- attribute1 <=> attribute2

The *Pre condition* is a constraint on the state of the sender to send the message. *Post condition* is a constraint on the state of the receiver to receive the message.

By specifying different Post conditions for the same message exchanged between the same two objects, we can have different actions depending on when the message is received. Other use of Post condition is to decide whether to accept or reject the message depending on its time of arrival.

For Example, when a student sends a registration form request to DPGC, depending on whether he had received Registration forms from DOAA or not, he will process the message or ignore it.

While coding, *Pre condition* is generally useful to position the function call.

Pre and Post conditions are also important for testing purposes.

Action : *action to be taken after this message has been received*

Action can be any of the following

- send message msg_id [of process_id]
- do sub_process sub_proc_id (parameters)
- send generic message msg_id {TO:value, FROM:value, ...etc.}
- do generic sub_process process_id (parameters) {values to substitute}
- select menu_item xxx

"select menu_item xxx" can be used only if the receiver is an Interface object.

generic message ::
declaration of normal message, except that some fields will be empty.

Comments specification

Finally, any text beginning with a # character upto the end-of-line is considered a comment in all specifications.

2.3 An example

2.3.1 Description of Doaa Office

DOAA Office deals with all the academic activities in the Institute. Registration of Students, Degree awarding, new Courses approval, Student's performance evaluation each semester, etc. Here we represent Registration process in detail.

Registration involves clearing the Institute dues, Hall dues, Academic registration and final registration. Academic registration for next semester is done before the end of the current semester. Dues clearing and final registration are done only during the specified dates, at the beginning of the semester.

For Academic registration, DOAA sends a notice giving the dates of registration. List of offered courses is put up on the notice boards. A student fills in a registration form available from the Convener, DUGC/DPGC depending on whether he/she is UG Student or PG Student. The student then selects his courses of interest from the list of courses offered (Courses may be compulsory or electives). Instructor selects some students depending on his selection procedure (for some elective courses the instructor's approval is required). After filling the registration form, student submits it to Convener, DUGC/DPGC and gets his signature.

Dues clearance is done by paying outstanding dues to the institute.

For final registration, Student submits Registration form to DOAA's office. DOAA office checks whether student has cleared dues or not, whether he is academically eligible for registration, and enters the Student's roll number on rolls. A Student is registered when his roll number is entered on rolls.

2.3.2 Requirements Specification

- *Registration Process*

DOAA sends registration notice to notice boards.
DOAA sends registration forms to Conveners, DUGC/DPGC.
Student requests DUGC/DPGC for registration form.
Student sees notice board for list of offered courses.

Student (UG) fills the registration form with the core courses and electives. For each course he selects, student requests course instructor for approval to credit that Course (if that course requires instructor's approval).
Student fills the Registration form and gets DUGC/DPGC's signature.
For clearing dues, the student pays the dues and gets receipt(s).
Finally, student submits the Registration form and these two receipts in the DOAA's Office.

After receiving the registration form, DOAA's office checks whether the student is academically eligible for registration or not. If he is eligible, then the student is deemed to have registered.

2.3.3 Initial Domain Object Model

Looking at description of the system, we find some obvious objects like STUDENT, COURSE, DOAA, DEPARTMENT, DPGC, DUGC, etc.
STUDENT will have attributes *name*, *address*, *roll_no*, etc. *name* and *address* are attributes for DOAA, DUGC, DPGC also. So, we can generalize these classes to form the PERSON class, whose attributes are *name*, *address*.

The complete object model is given below.

2.3.4 Object Model

- *Class Specifications*

```
#-------------------------------------------------------------------
CLASS : PERSON.
TYPE : ABSTRACT.
INHERITS : PERSON.
GENERALISATION OF : STUDENT, FACULTY_MEMBER.
ATTR : name <NAME>,
       email_addr <EMAIL>,
       address <STRING>.
#-------------------------------------------------------------------
```

```
CLASS : STUDENT.
TYPE : ABSTRACT.
INHERITS : PERSON.
GENERALISATION OF : UG_STUDENT, PG_STUDENT.
ATTR : roll_no <NUMBER>,
       address <STRING>,
       joining_date   <DATE>, #date of joining the institute
       semester  < NUMBER >. # Academic semester for student
RELATED WITH : COURSE (takes, N),
               PERFORMANCE_RECORD (has, 1).
#-------------------------------------------------------------------
CLASS : UG_STUDENT.
INHERITS : STUDENT.
ATTR : acad_status < STRING >, # regular/warning/probation .. etc.
RELATED WITH : DUGC (CONVENER, 1).
#-------------------------------------------------------------------
CLASS : PG_STUDENT.
TYPE : ABSTRACT.
INHERITS : STUDENT.
GENERALISATION OF : MTECH_STUDENT, PHD_STUDENT.
ATTR : leave_status { casual_leaves <NUMBER>, sick_leaves <NUMBER>,
                      vacation_leaves <NUMBER> }
#-------------------------------------------------------------------
CLASS : MTECH_STUDENT.
INHERITS : PG_STUDENT.
RELATED WITH : DPGC (CONVENER, 1),
               THESIS_SUPERVISOR (Th_guide, 1),
               TA_SUPERVISOR (Ta_instr, 1).
#-------------------------------------------------------------------
CLASS : PHD_STUDENT.
INHERITS : PG_STUDENT.
RELATED WITH : DPGC (CONVENER, 1),
               THESIS_SUPERVISOR (Th_guide, 1),
               PHD_ACAD_DETAILS (acad_details, 1).
#-------------------------------------------------------------------
CLASS : REG_FORM.
ATTR : year< number>,
       sem< number>,
       roll_no< number>,
       room_no  < string >,
       student_name<STRING>,
       Thesis_supervisor<STRING>,
       Financial_Asst <string>,
       DPGC_sign<signature>,
       DUGC_sign  <signature>,
```

```
       courses < TABLE [ course_no <NUMBER>, c_name <STRING>,
                         units <number>, instructor <string> ] >.
#-------------------------------------------------------------------
CLASS : FACULTY_MEMBER.
INHERITS : PERSON.
GENERALISATION OF : COURSE_COORDINATOR, CONVENER, DOAA.
ATTR : pf_no <NUMBER>,
       leave_status { casual_leaves <NUMBER>, sick_leaves <NUMBER>,
                      vacation_leaves <NUMBER>}.
#-------------------------------------------------------------------
CLASS : COURSE.
ATTR :
       c_no <number>,
       c_name <string>,
       units_pg <number>,
       units_ug  <number>,
       syllabus <string>,
       references  <string>,
       approval_date  <date>,
       lab <number>,
       lecture <number>,
       tutorial  <number>,
       course_slots < TABLE [ time_slot<time>,day<string>,room_no<string>]

RELATED WITH : COURSE_COORDINATOR (taught_by, N),
               STUDENT (enrolled_by, N).
#-------------------------------------------------------------------
CLASS : COURSE_COORDINATOR.
INHERITS : FACULTY_MEMBER.
RELATED WITH : COURSE (teaches, N).
#-------------------------------------------------------------------
CLASS : CONVENER.
TYPE : ABSTRACT.
GENERALIZATION OF : DPGC, DUGC.
RELATED_WITH : RULE_BOOK (Rules_store, 1).
#-------------------------------------------------------------------
CLASS : DPGC.

INHERITS : CONVENER.
#-------------------------------------------------------------------
CLASS : DUGC.

INHERITS : CONVENER.

#-------------------------------------------------------------------
```

```
CLASS : DOAA.
INHERITS : FACULTY_MEMBER.
RELATED WITH : DPGC (DPGC, 1),
               DUGC (DUGC, 1),
               RULE_BOOK (Rules_store, 1).
#-------------------------------------------------------------------
CLASS : NOTICE_BOARD.
AGGREGATION OF : NOTICE (N).
#-------------------------------------------------------------------
CLASS : NOTICE.
ATTR : date <DATE>,
       issued_by <STRING>,
       title <STRING>,
       text < TEXT>.
#-------------------------------------------------------------------
CLASS : RULE_BOOK.
RELATED WITH : RULE (has, N).
#-------------------------------------------------------------------
CLASS : RULE.
ATTR : rule_id <STRING>,
       w_e_f <DATE>,
       eff_upto <DATE>,
       rule_func <FUNCTION>.
#-------------------------------------------------------------------
CLASS : TIME_TABLE.
ATTR : time_table < TABLE [ c_no <NUMBER>, Instructor <STRING>, schedule
                          <TABLE [ day <STRING>, time_slot <[TIME]>] > ] >.
RELATED WITH              : DEPARTMENT ( Belongs_to , 1) .
#-------------------------------------------------------------------

CLASS                     : DEPARTMENT.
RELATED WITH              :
        STUDENT ( Has , 1-N) ,
        FACULTY ( Has , M-N) ,
        NOTICE_BOARD ( Has , 1-1) ,
        TIME_TABLE (Has, 1-1).

#-------------------------------------------------------------------

CLASS : ACCOUNTS_SECTION.

INHERITS : GENERAL_ADMINISTRATOR.

#-------------------------------------------------------------------
```

```
CLASS : HALL_OFFICE.

INHERITS : GENERAL_ADMINISTRATOR.

#-------------------------------------------------------------------
CLASS : CALENDAR.
ATTR : calendar <TABLE [ date <DATE>, event <CALENDAR_EVENT_TYPE>,
                         semester <STRING>] >,
       current_sem <SEMESTER>,
       current_year <YEAR>.
#-------------------------------------------------------------------
```

Interface Objects specification

```
#-------------------------------------------------------------------

CLASS            : STUDENT_INTERFACE.
TYPE             : INTERFACE
RELATED WITH     : A_D_FORM_INTERFACE ("uses", 1-1),
                   STUDENT ("attached to", 1-1).
#-------------------------------------------------------------------

CLASS   : REG_FORM_INTERFACE.
TYPE    : INTERFACE.
RELATED : REG_FORM
#-------------------------------------------------------------------

CLASS    :        DOAA_INTERFACE.
TYPE     :        INTERFACE.
RELATED WITH : DOAA.
#-------------------------------------------------------------------

CLASS            : COURSE_COORD_INTERFACE.
TYPE             : INTERFACE.
RELATED WITH : COURSE_COORDINATOR(coord,1).
#-------------------------------------------------------------------

CLASS   : NOTICE_INTERFACE.
TYPE    : INTERFACE
RELATED WITH     : NOTICE (notice, 1).

#-------------------------------------------------------------------
```

2.3.5 Message Model

- *Registration process*

```
process :: Registration.
#----------------------#

FROM        : DOAA_INTERFACE.
TO          : SELF.
msg_id      : prepare_reg_notice ().
Action      : do generic sub_process make_notice {FROM : $FROM,
                                    res_message : reg_notice}
result      : reg_notice <NOTICE>.

FROM        : DOAA_INTERFACE.
TO          : DOAA.
msg_id      : send_reg_notice_to_depts (reg_notice <NOTICE>).
PRE_COND    : received resultof_prepare_reg_notice.
Action      : send message notice_for_registration.

FROM        : DOAA.
TO          : CONVENER (ALL).
msg_id      : notice_for_registration (reg_notice <NOTICE>).
Action      : do sub_process store_notice.

FROM        : STUDENT_INTERFACE.
TO          : STUDENT.
msg_id      : get_reg_form ().
Action      : send message req_for_reg_form {FROM:STUDENT, TO:CONVENER}

FROM        : STUDENT.
TO          : CONVENER.
msg_id      : req_for_reg_form ().
result      : res_val {reg_form <REGISTRATION_FORM>, remarks <STRING>}.
Action      : do sub_process check_eligibility_for_reg.

# sending message to abstract class 'CONVENER' means sending to one of the
# derived classes.

FROM        : STUDENT.
TO          : REGISTRATION_FORM.
msg_id      : fill_form (values <STRING>).
msg_type    : exclusive. # this type takes care of security.

FROM        : STUDENT.
TO          : CONVENER.
```

```
msg_id     : sign_reg_form (reg_form <REGISTRATION_FORM>).
result     : reg_form <REGISTRATION_FORM>. # signed registration form.
Action     : do sub_process check_and_sign_reg_form
                            (reg_form <REGISTRATION_FORM>).

FROM       : STUDENT.
TO         : DOAA.
msg_id     : req_for_reg (reg_form <REGISTRATION_FORM>).
result     : remarks <STRING>.
Action     : do sub_process process_the_reg_form.

#-------------------------------------------------------------------------#

sub process :: process_the_reg_form.
#---------------------------------#

FROM       : DOAA.
TO         : SELF.
msg_id     : extract_details (query = "roll_no" <STRING>, reg_form
                              <REGISTRATION_FORM>).
             # extract_details should be taken as a general purpose message
result     : roll_no <NUMBER>.

FROM       : DOAA.
TO         : HALL_OFFICE.
msg_id     : check_student_for_dues (roll_no <NUMBER>).
result     : paid <BOOLEAN>.

if (paid == TRUE) {

    FROM       : DOAA.
    TO         : ACCOUNT_SECTION.
    msg_id     : check_student_for_dues (roll_no <NUMBER>).
    result     : paid_instt_dues <BOOLEAN>.

    if (paid_instt_dues == TRUE) {

        FROM       : DOAA.
        TO         : SELF.
        msg_id     : form_links ().
        Action     : do sub_process form_assoc_between_stud_course
                                    ( reg_form <REGISTRATION_FORM>) .

    } else {
```

```
        FROM        : DOAA.
        TO          : STUDENT (BACK).
        msg_id      : replyto_req_for_reg (remark = "Instt. dues not paid"
                                           <STRING>).
        msg_type    : reply.

    }

} else {

    FROM        : DOAA.
    TO          : STUDENT (BACK).
    msg_id      : replyto_req_for_reg (remark = "Hall duejs not paid" <STRING>).
    msg_type    : reply.

}

#-------------------------------------------------------------------------------#

sub process :: form_association_between_student_course.
#-----------------------------------------------------#

FROM        : DOAA.
TO          : SELF.
msg_id      : extract_details (query = "roll_no, courses" <STRING>,
                               reg_form <REGISTRATION_FORM>).
result      : {roll_no <NUMBER>, courses <TABLE [c_no <STRING>, c_name <STRING>,
                                           units <NUMBER>, instructor <STRING>]>}.

foreach c_no (courses) {

    FROM        : DOAA.
    TO          : COURSE (c_no == COURSE::c_no).
    msg_id      : give_handle ().
    msg_type    : shared.
    result      : course_ref <HANDLE>.

    FROM        : DOAA.
    TO          : STUDENT (roll_no == STUDENT::roll_no).
    msg_id      : give_handle ().
    msg_type    : shared. # for security
    result      : student_ref <HANDLE>.

    FROM        : DOAA.
    TO          : STUDENT (roll_no == STUDENT::roll_no).
```

```
    msg_id     : form_assoc_with_course (course_ref <REFERENCE>).
    msg_type   : exclusive.

    FROM       : DOAA.
    TO         : COURSE (c_no == COURSE::c_no).
    msg_id     : form_assoc_with_student (student_ref <REFERENCE>).
    msg_type   : exclusive.

}

#-------------------------------------------------------------------------#


sub process :: change_the_reg_form.
#---------------------------------#

 FROM                  : STUDENT
 TO                    : SELF
 msg_id                : retrive_reg_form ( constraints <STRING> )
 result                : reg_form <REG_FORM>
 Action                : send message modify_attr.


 FROM                  : STUDENT
 TO                    : SELF
 msg_id                : modify_attr ( reg_form <REG_FORM>,
                                       values_string <STRING> )

#-------------------------------------------------------------------------#

sub process :: check_and_sign_reg_form.
#-------------------------------------#

FROM       : CONVENER.
TO         : SELF.
msg_id     : check_reg_form (reg_form <REGISTRATION_FORM>).
result     : correct <BOOLEAN>.

if (correct == FALSE) {

    FROM       : CONVENER.
    TO         : STUDENT (BACK).
    msg_id     : replyto_sign_reg_form (remarks = "Incorrect" <STRING>).
    msg_type   : reply.
```

```
} else {

    FROM       : CONVENER.
    TO         : REGISTRATION_FORM.
    msg_id     : put_dpgc_sign (signature <SIGNATURE>).
    msg_type   : exclusive.

}

#-------------------------------------------------------------------------------#


sub process :: check_eligibility_for_reg.
#--------------------------------------#

FROM       : CONVENER.
TO         : SELF.
msg_id     : check_for_reg_notice ().
result     : present <BOOLEAN>.

if (received message reg_notice of Registration) {

    FROM       : CONVENER.
    TO         : STUDENT (BACK).
    msg_id     : give_details ("status,desgn" <STRING>).
    result     : details <STRING>.

    FROM       : STUDENT.
    TO         : CONVENER (BACK).
    msg_id     : replyto_give_details (details <STRING>).

    FROM       : CONVENER.
    TO         : RULE_BOOK.
    msg_id     : give_rule ("reg_elig_check_rule" <STRING>, desgn <STRING>).
    result     : rule <RULE>.

    FROM       : CONVENER.
    TO         : SELF.
    msg_id     : apply_rule (rule <RULE>, details <STRING>).
    result     : eligibile <BOOLEAN>.

    if (eligible == TRUE) {

        FROM       : CONVENER.
        TO         : SELF.
```

```
        msg_id      : create_reg_form ().
        result      : reg_form <REGISTRATION_FORM>.

        FROM        : CONVENER.
        TO          : STUDENT (BACK).
        msg_id      : replyto_req_for_reg_form ( reg_form <REGISTRATION_FORM>,
                              remarks = "Eligible for Registration" <STRING> ).
        msg_type    : reply.
        Action      : do store (reg_form <REGISTRATION_FORM>),
                      do store_temp (remarks <STRING>).

    } else {  # eligible is FALSE

        FROM        : CONVENER.
        TO          : STUDENT (BACK).
        msg_id      : replyto_req_for_reg_form (reg_form <REGISTRATION_FORM>,
                          remarks = "Not eligible for registration" <STRING>).
                      # reg_form will be NULL.

        msg_type    : reply.
        Action      : do store (reg_form <REGISTRATION_FORM>),
                      do store_temp (remarks <STRING>).

    }

} else { # Registration notice has not been received from DOAA.
         # means it is not yet time for registration

    FROM        : CONVENER.
    TO          : STUDENT (BACK).
    msg_id      : replyto_req_for_reg_form (reg_form <REGISTRATION_FORM>,
                        remarks = "not time for registration" <STRING>).
    msg_type    : reply.
    Action      : do store (reg_form <REGISTRATION_FORM>),
                  do store_temp (remarks <STRING>).

}

#-------------------------------------------------------------------------#
```

Generic processes

```
generic sub process :: make_notice.

INPUTS : ISSUER.
#---------------------------------#

FROM       : ISSUER
TO         : NOTICE_INTERFACE.
msg_id     : make_notice ().
result     : res_notice <NOTICE>.

FROM       : NOTICE_INTERFACE.
TO         : ISSUER.
msg_id     : replyto_make_notice (notice <NOTICE>).
msg_type   : reply.

#-------------------------------------------------------------------------------#

generic sub process :: get_approval_for_leave.
#-----------------------------------------#
INPUTS  :: SUPERVISOR.


FROM       : STUDENT.
TO         : SELF.
msg_id     : get_Supervisor (  query_str = SUPERVISOR <STRING> ).
result     : supervisor <FACULTY_MEMBER>.

FROM       : STUDENT.
TO         : SUPERVISOR.
msg_id     : req_for_leave_approval (  ).
Action     : do sub_process store_request.

#-------------------------------------------------------------------------------#
```

Chapter 3

Translation

In this chapter we describe the implementation phase and discuss how to translate each element of the specification into C++ code. The programming language C++ contains features like data encapsulation, derivation, virtual functions, late binding and template mechanism, which make the implementation of general OO concepts like data-hiding, generalization, polymorphism, meta data etc., easy. There is no feature in C++ that can implement association and aggregation directly. We discuss a suitable mechanism to implement association and aggregation.

The Message Based Specification System gives Object and Message models of the system. In the following sections we explain how each statement of these models can be mapped to C++ code.

3.1 Object Model

The Object Model gives the static structure of the system. The classes presented in the system, their attributes and the relationship between the classes are specified in the Object Model.

3.1.1 Class and It's Attributes

A class of this methodology is directly mapped to a C++ class. The attributes specified in this class become the data members of the C++ class. The default

accessibility of these data members is private. The accessibility of the data members of a generalized class is protected. Here we assume that all the attributes of the generalized class are shared by the specialized classes. The generalization is implemented in C++ using class derivation.

We provide a persistent class. The derived classes of this persistent class are also persistent. For more information about the maintenance of persistent of the objects please see the Appendix A.

Eg:

Specification for Student Class

```
CLASS : STUDENT.
TYPE : ABSTRACT.
INHERITS : PERSON.
ATTR : roll_no <NUMBER>,
       name <STRING>,
       joining_date   <DATE>, #date of joining the institute
       semester  < NUMBER >.  # Academic semester for student
```

Translation of Student Class in C++ :

```
class  STUDENT : public PERSON {
public:
private:
   NUMBER  roll_no ;
   STRING  name ;
   DATE    joining_date ;
   NUMBER  semester  ;
};
```

3.1.2 Association

In the specification the association is represented by *RELATED WITH* field of a class specification. The concept of association has no direct counterpart in C++. The Association between two objects can be implemented by using pointers. But the objects are persistent, so an object cannot have pointers to related objects residing on the disk. So, we have to use object-ids to maintain the association between objects. In each object, there should be a data member whose value can identify this object uniquely in the inheritance hierarchy of this object. We are using such object-id as such a data member. The mechanism that implements the association should ensure referential integrity between the linked objects. The association is always bidirectional. Each object will have a link to the related object. The link can be traversed in both directions.

In the implementation of association between objects, each object will have a list of object-ids of all related objects. When an object wants to send a message to another object, it has to get a pointer to the receiver object. The sender object uses the object-id of the receiver object to get a reference to the receiver object.

For each association, in each related class there will be one *private object member* and two private methods, *set* and *unset*. This special object maintains the object-ids of the related objects. The methods *set* and *unset* are used to add and delete respectively an object-id of the related object . When a link is created/deleted the cardinality of the association will be checked. If the cardinality constraints are not met an error will be reported. The functions

- CreateLink
- DeleteLink

are used for linking and unlinking the objects.

The function *CreateLink(Object A, Object B)* forms a link between objects A and B. The function *DeleteLink(Object A, Object B)* deletes a link between the objects A and B. These two functions are friend functions to both objects. These functions invoke the private methods, *set* and *unset*. Thus a link cannot be destroyed or

formed without using these functions. Using these functions to create and destroy the links ensures the referential integrity between related objects.

Eg:

Classes Student and Course are related to each other. The specification of these classes is

```
CLASS : STUDENT.
ATTR  : . . .
RELATED WITH : COURSE(myCourses, N);

CLASS : COURSE.
ATTR  : . . .
RELATED WITH : STUDENT(creditedby, N);
```

Translation of Student Class in C++ :

```
class  STUDENT {
  friend CreateLink(STUDENT *, COURSE *);
  friend DeleteLink(STUDENT *, COURSE *);
public:
private:

   // other attributes

  KeysObject myCoursesKeys; // maintains object-ids
  void set(COURSE *c) {
      myCoursesKeys.Add( c -> GetKeyField() );
  }
  void unset(COURSE *c) {
      myCoursesKeys.Remove( c -> GetKeyField() );
  }
};
```

Translation of Course Class in C++ :

```
class  COURSE {
  friend CreateLink(STUDENT *, COURSE *);
  friend DeleteLink(STUDENT *, COURSE *);
public:
private:

  // other attributes

  KeysObject creditedbyKeys;
  void set(STUDENT *s) {
      creditedbyKeys.Add( s -> GetKeyField() );
  }
  unset(STUDENT *s) {
      creditedbyKeys.Remove( s -> GetKeyField() );
  }
};
```

3.1.3 Aggregation

Aggregation stands for a kind of part_of relationship between objects. Aggregation is a special kind of association. Aggregation is implemented in the same way as association. Whenever an object wants to send any message to it's constituent object, it uses the object-id to get a pointer to the part_of or constituent object.

Eg:

A Window is an aggregation of Borders. Specification is

```
CLASS : WINDOW .
AGGREGATION OF : BORDER(4);
```

Generated Code is :

```
class WINDOW {
public :
void AddBORDER(BORDER *b) {
        set(b);
        b -> Save();
}
List<BORDER *>  *GetBORDERS() {

        List<BORDER *> *l = new List<BORDER *>;
        // get pointers to the related objects
        GetPtrsToRelatedObjects(theBORDERS, l);
        return l;
}
private :
   KeysObject  theBORDERS;
   void set(BORDER *b) {
        if(b) theBORDERS.Add ( b -> GetKeyField() );
   }
   void unset(BORDER *b) {
        if(b) theBORDERS.Remove ( b -> GetKeyField() );
   }
};
```

3.2 Message Model

In the *Message Model* messages exchanged by system objects are specified. For each object of the system, we pick up the methods from the given message scenarios. The objects are viewed as servers responding to requests via message passing. Every message is a call to an operation or a return. An object can communicate with

an independent object if it knows it's name or object identifier. In response to a message an object executes a sequence of primitive actions

- Inquiring/Updating the local memory of the object.
- Create new objects
- Sending messages and receiving replies and
- Return values as reply to the message

In the following examples we explain how an object can send/receive a message to/from another object and to/from itself, how it gets a reference to the receiver object, and how it performs the actions specified in the message.

3.2.1 Methods of a Class

An object's behavior cannot be specified in the *Object Model*, since it is a result of all the interactions the object has with other objects. This is captured by the *Message Model*. The basic idea which implements the sending and receiving of a message is: *When an object A wants to send a message M to another object B, the object A invokes the method of B that corresponds to the receiving of the message M.*

For each message specified in the *Message Model*, the sender object will have a method that carries out task of sending the message, and the receiver object will have a method that carries out the task of receiving a message. Every method that corresponds to sending a message is an inline method containing code for checking the pre-conditions and a call to the method of the receiver object that corresponds to receiving the message. If the pre-conditions specified fail, it does not invoke the receiving method of the receiver object. After testing of the system, we can remove the sending method along with the pre-conditions, and directly call the receiver object's receiving message method.

Accessibility of the Sending and Receiving Methods

For a concrete class all the sending methods are private and all the receiving methods are public. For an abstract class all the sending and receiving methods are protected. For interface classes all the sending and receiving methods are public.

An abstract class does not have any instances. So it neither receive nor send any message. But in the Message Model an abstract class also send/receive some messages. In such cases the actual sender/receiver will be any one of the sub classes and is known only at run time. This is possible in C++ as it provides dynamic binding and virtual functions.

Parameters of the Sending and Receiving Methods

The arguments for sending method are

- A pointer to the receiving object and
- Parameters of the corresponding message.

The arguments for receiving method are

- A pointer to the sender object and
- Parameters of the corresponding message.

3.3 Implementation of various types of messages

The messages exchanged between the objects are classified into two different types.

- Normal message
- Reply message

3.3.1 Implementation of Normal message

An object may receive a normal message from an object of any type. The sender sends it's reference along with the message parameters. It may be needed to communicate with the sender while servicing the request.

Eg:1

Suppose class A receives message M with parameter p1 and p2 of type int, from class B. And class A sends message N to the class B.

Generated code will be.

```
class A  {
public :
  rcv_M(B *sender, int p1, int p2);
private :
  send_N(B *rcvr) { rcvr -> rcv_N(this); }
};
```

```
class B {
public :
  rcv_N(A *sender);
private :
  send_M(A *rcvr, int p1, int p2) { rcvr-> rcv_M(this, p1, p2); }
};
```

Eg:2

Suppose class A receives message M with parameter p1 and p2 of type int, from classes B and C. And class A sends message N to these classes(B, C).

```
class A {
public :
  rcv_M(AbstractClass *sender, int p1, int p2);
private :
  send_N(AbstractClass *rcvr) { rcvr -> rcv_N(this); }
};

class AbstractClass {
public :
   virtual rcv_N(A *sender) {}; // dummy method
};

class B : virtual public AbstractClass {
public :
  rcv_N(A *sender);
private :
  send_M(A *rcvr, int p1, int p2) { rcvr -> rcv_M(this, p1, p2); \}
}

class C : virtual public AbstractClass {
public:
  rcv_N(A *sender);
private :
  send_M(A *rcvr, int p1, int p2) { rcvr -> rcv_M(this, p1, p2); }
};
```

3.3.2 Implementation of Reply message

The reply message is sent in response to a message. This is not implemented as a function call. The reply message is mapped to a return statement in the definition

of the method. The message parameters will become the return values. *Eg:*

```
FROM        : CONVENER.
TO          : STUDENT
msg_id      : give_details ();
result      : details <STRING>.

FROM        : STUDENT.
TO          : CONVENER (BACK).
msg_id      : replyto_give_details (details <STRING>).
```

Code of the receiver method of STUDENT for receiving the message *give_details.*

```
STRING STUDENT :: R_give_details(CONVENER *sender)
 {

     STRING details;
     // construct the details
     return ( details );
 }
```

3.4 Actions

When an object receives a message, it may have to make some other calls to service the request of the sender object. What the receiver object has to do after reception of a message, is specified in the *ACTION* field of the message specification. There are precisely two types of actions.

1. Send a message
2. Do a sub process

The action *Send a message* is translated to a function call in the definition of the receiving method of the receiver. Here the pre-condition of type *received some message* is useful in identifying the correct message to be sent.

Eg:1

When COURSE_COORDINATOR receives the message *get_registered_students* from COURSE_COORD_INTERFACE it does the action *send message give_registered_students.* The specification is

```
FROM       : COURSE_COORD_INTERFACE.
TO         : COURSE_COORDINATOR.
msg_id     : get_registered_students ( course_no <STRING>).
result     : registered_students<TABLE[student<NAME>,roll_no<NUMBER>]>.
Action     : send message give_registered_students.

FROM       : COURSE_COORDINATOR.
TO         : COURSE (c_no = course_no).
msg_id     : give_registered_students ().
result     : registered_students<TABLE[student<NAME>,roll_no<NUMBER>]>.
PRE_COND   : received get_registered_students.
```

The definition of the method that corresponds to the receiving of the message *get_registered_students* in COURSE_COORDINATOR is

```
COURSE_COORDINATOR::R_get_registered_students(COURSE_COORD_INTERFACE *sender,
                                              STRING course_no
                                             )
{
     /* performing the specified  action
        send message give_registered_students */

    // get pointer to the receiver
    COURSE *receiver;
```

```
        GetPtrToObject(receiver, course_no);
        registered_students = S_give_registered_students(receiver);
        return(registered_students);
    }
```

The action *Do sub process* is translated to a statement that invokes a private method that corresponds to this sub process.

Eg:2

In the Registration process when DOAA receives the message *req_for_reg* from STUDENT, it does the action *do sub_process process_the_reg_form*. The specification is

```
FROM        : STUDENT.
TO          : DOAA.
msg_id      : req_for_reg (reg_form <REGISTRATION_FORM>).
result      : remarks <STRING>.
Action      : do sub_process process_the_reg_form.
```

The definition of the method that corresponds to the receiving of the message *req_for_reg* in DOAA is

```
STRING DOAA :: R_req_for_reg ( STUDENT *sender,
                               REGISTRATION_FROM  reg_form
                             )
 {
   // Rest of the code to be filled.
   // Make a call to private method (sub_process) process_the_reg_form.
       process_the_reg_form();
   // Data to be sent to the caller of this method (Result).
      STRING remarks;
      return (remarks);
 }
```

How to get a reference to a receiving object

Before sending a message to an object, the sender must identify and get a reference to the legal receiver object. The receiver object can be

- a globally known object
- the sender itself may become the receiver.
- one of the related objects
- some other object

At any instance the receiver object may be in the memory (active object) or on the disk (passive object). We provide a routine which gives the pointer to an object given its object-id(key). This routine gives a pointer to the active object. If the required object is not active, it is made active and pointer to it is returned.

Use of Receiver Constraints :-

In the specification the type of the receiver object and the receiver constraints are given in the *TO* field of the message specification.

If the constraint is *ALL*, the sender object sends the message to all the objects of specified type present in the system. We provide a routine that returns pointers to all objects of a specific class.

Eg:3

```
FROM       : DOAA_INTERFACE.
TO         : DOAA.
msg_id     : send_reg_notice_to_depts (reg_notice <NOTICE>).
Action     : send message notice_for_registration.

FROM       : DOAA.
TO         : CONVENER (ALL).
msg_id     : notice_for_registration (reg_notice <NOTICE >).
Action     : do sub_process store_notice.
```

The generated code of the method of DOAA corresponding to the message *send_reg_notice_to_depts* received from DOAA_INTERFACE is

```
int DOAA :: R_send_reg_notice_to_depts(DOAA_INTERFACE *sender,
                                       NOTICE reg_notice
                                      )
  {
      List<CONVENER *>  *rcvrList; // list of pointers to CONVENERs
      // get the list of ptrs to CONVENERs
      GetPtrsToObjects(rcvrList);
      if(rcvrList == NULL) {
           cout << '' NO OBJECTS OF TYPE CONVENER \n'';
           return -1;
      }
      // send message to all CONVENERs
      Node<CONVENER *> *header = rcvrList->GetHeader();// list header
      while(header != NULL) {
        CONVENER *rcvr = header -> GetItem();// receiver pointer
        S_notice_for_registration(rcvr, reg_notice);// sending message
        header = header -> GetNext();
      }
      return 0;
  }
```

If the receiver constraint is *ALL_RELATED* the sender sends the message to all lated objects of the specified type. The related objects are just associated objects. ach object maintains object-ids(keys) of it's related objects.

Eg:4

Suppose COURSE gets a request to give all the roll numbers of the students crediting that Course. Then COURSE has to get roll numbers from all related students.

```
FROM        : COURSE_INTERFACE.
TO          : COURSE.
msg_id      : give_credited_students_rolls().
result      : roll_numbers <[NUMBER]>.  #list of numbers
Action      : send message give_roll.

FROM        : COURSE.
TO          : STUDENT (ALL_RELATED).  # All STUDENT objects Related
                                      # COURSE object.
msg_id      : give_roll_number();
result      : roll_no <NUMBER>.
PRE_COND    : received message give_credited_students.
```

The generated code for the method of COURSE corresponding to the message *give_credited_students_rolls* received from COURSE_INTERFACE is

```
List<NUMBER>    *COURSE ::
R_give_credited_students_rolls(COURSE_INTERFACE *sender)
{
    List<NUMBER>  *result = NULL ; // result

    List<STUDENT *> *STUDENTlist; // list of ptrs to STUDENTs
    // get list of ptrs to STUDENTs
    // STUDENTkeys is a member object that maintains keys of STUDENTs.
    GetPtrsToRelatedObjects( STUDENTkeys, STUDENTlist );
```

```
    // sending message to all STUDENTs
    Node<STUDENT *>  *header = NULL; // list header
    if(STUDENTlist)  header = STUDENTlist -> GetHeader();
    while(header) {
        STUDENT *recvr = header -> GetItem(); // receiver pointer
        NUMBER  resultListitem; // result expected
        resultListitem = S_give_roll_number(recvr); // sending message
        result.Append(resultListitem); // append to the result list
        header = header -> GetNext();
    }
    return result; // send back result
}
```

Sometimes the receiver has to send a message to the sender of some previous message it had received. The receiver constraint *BACK* is used to specify this situation

Eg:5

When a STUDENT sends message req_for_reg_form to CONVENER, the CONVENER sends back the message give_status to the student.

```
FROM       : STUDENT.
TO         : CONVENER.
msg_id     : req_for_reg_form ().
result     : res_val reg_form <REGISTRATION_FORM>.
Action     : send give_status.

FROM       : CONVENER.
TO         : STUDENT (BACK).
msg_id     : give_status();
result     : status <STRING>.
PRE_COND   : received message req_for_reg_form .
```

The generated code is

```
REGISTRATION_FORM  CONVENER :: R_req_reg_from ( STUDENT *sender )
  {
       STRING status; // result expected
       status = S_give_status(sender);  // sending a message back
  }
```

If the constraint is some *Condition* then the sender object has to send the message to all the objects of specified type, that satisfy the *Condition*. The sender gets a list of pointers to the objects of specified type using the routine *GetPtrsToObjects*.

If there is no receiver constraint and if the sender class is related to the receiver class with cardinality one, then it is assumed that this related object is the receiver object. Each object has a key to its related object. It gets the pointer to the related object using the routine *GetPtrToObject* and sends a message to it.

If there is no receiver constraint, the following cases are taken as specification mistakes.

- if there is more than one related object of receiver type
- if the receiver object is un-related to the sender object.

3.5 Sub Process

A sub process is a group of interactions of an object with other objects. Finally each sub process is converted to a private method of the object that initiates the sub process. The result of the terminating message of the sub process becomes the return value of this method.

Eg:6

```
sub process :: check_and_sign_reg_form.

   FROM       : DPGC.
   TO         : SELF.
   msg_id     : check_reg_form (reg_form <REGISTRATION_FORM>).
   result     : correct <BOOLEAN>.

   if (correct == FALSE) {

    FROM       : DPGC.
    TO         : STUDENT (BACK).
    msg_id     : replyto_sign_reg_form (remarks <STRING>).
    msg_type   : reply.

   } else {

      FROM       : DPGC.
      TO         : REGISTRATION_FORM.
      msg_id     : put_dpgc_sign (signature <SIGNATURE>).
    }
```

This sub_process is initiated by DGPC . This sub_process is mapped to a private method of DPGC .

```
STRING  DPGC :: check_and_sign_reg_form(REGISTRATION_FORM  reg_form) {
  // select parameters
     REGISTRATION FORM  reg_form ;
  // This is Private Method
  BOOLEAN  condition; // result
  condition = check_reg_form(reg_form); //sending the message
  if(condition == FALSE ) {
```

```
  // sending result
   STRING remarks;
   return ( remarks );
  }
  else {
    // select parameters
    SIGNATURE signature;
    // select the receiver
    REGISTRATION_FORM  recvr;
    //send the message
    S_put_dpgc_sign(recvr, signature) ;
  }
}
```

Chapter 4

Translator

4.1 Implementation

The implementation of the translator consists of, mainly, two stages

- Parsing
- Conversion

4.1.1 Parsing

Tools, *Bison* and *Flex* are used to construct the parser. Bison is a parser generator and Flex generates lexical analyzers.

The object model consists of all information on classes. A class-table is constructed while parsing the object model. The class-table consists of information on all the classes specified in the object model. Generic-message and generic-sub-process tables are constructed while parsing the generic messages and generic sub-processes. These tables are used in expansion of the generic messages and generic sub-processes. The generic messages/sub-processes are expanded wherever they are encountered while parsing processes and sub-processes. The information on the processes and sub-process is stored in process-table and sub-process table respectively.

An example for expansion of a generic message : Specification of a generic message *reg_notice* is

```
generic message ::
FROM       : DOAA.
TO         : * .
msg_id     : reg_notice (notice_str <STRING>).
Action     : do sub_proc store (notice_str <STRING>).
```

Specification of message *send_reg_notice_to_depts* that uses the generic message *reg_notice* is

```
FROM       : DOAA_INTERFACE.
TO         : DOAA.
msg_id     : send_reg_notice_to_depts ().
Action     : send generic message reg_notice {TO:DPGC, TO_constraint:(ALL)}
```

After generic message expansion specification will be

```
FROM       : DOAA_INTERFACE.
TO         : DOAA.
msg_id     : send_reg_notice_to_depts ().
Action     : send message reg_notice

FROM       : DOAA.
TO         : DPGC(ALL) .
msg_id     : reg_notice(notice_str <STRING>).
PRE_COND   : received message send_reg_notice_to_depts.
Action     : do sub_proc store(notice_str <STRING>).
```

4.1.2 Conversion

The conversion process follows the mapping rules described in the previous chapter. For each class, all the sending and receiving messages are extracted from the process and sub-process tables. The class-table, sending message and receiving message tables are used for generating class declarations and possible code for methods of the classes.

4.2 Inputs and Outputs

Input

Input is expected in a file in the following order.

- classes
- generic messages
- generic sub-processes
- sub-processes
- processes

Output

Output of this translator consists of declarations of classes, definitions of methods of the classes, errors and warnings. Declaration of a class X is given in a file with name "X.h" and methods of this class in a file "X.c". All errors and warnings are given in a file "errs_warns"

4.3 Usage

A file containing the specification is given to the translator :

<prompt> translator-name <input-file>

Chapter 5

Conclusions

5.1 Conclusions

In the previous chapters we explained how a message centric analysis/design specification can be translated into an implementation in C++. The translator developed will translate an analysis/design specification into C++ class headers and will generate possible code for methods of the classes. Associations and aggregations are implemented properly. A library has been developed that supports persistence of objects and provide routines which return pointers to required objects present in the system (appendices A & B). Code generated is reliable as it is generated under the specified design constraints.

Translation of the analysis/design specification into an implementation in RDBMS is described in [Vij96].

5.2 Exceptions

- We assumed that all objects of the system reside on local machine. It does not generate code to send/receive a message to/from an object on a remote machine.
- Code generated does not handle concurrency between objects.

- An object should have its own *read* and *write* methods to support persistence of the object (see appendix A). The definitions of these methods depend on type of each data member of the object. If an object contains data members of user defined type(date, table, list etc,.), the generated read and write methods have to be modified to handle such user defined data types.

- The key words of the specification language should not be used as names of classes, messages, attributes, or parameters.

5.3 Future Work

The specification language can be improved to represent an object oriented distributed system. This work can be extended to generate code which provides object location transparency and handles concurrency between objects.

Appendix A

Persistence Library

The OO language C++ does not support the persistent objects. We have to use a library that provides persistent objects. We developed a library that uses the ndbm.h library, a library available in unix, to store and retrieve the objects. We developed a persistent class which makes its derived classes persistent.

A.1 Object Id

Each object should have a key field that can identify this object uniquely in the inheritance hierarchy of this object. And a method that returns the object key is to be defined in each object class.

A.2 Persistent Class

The Persistent Class is an abstraction for all Persistent Objects which inherit from it for persistence support. The persistent class provides methods to save, retrieve and remove an object from the disk. The Persistent Class expects the following methods to be overridden in the class that wants to be persistent.

- GetInfo()
- Write(ostrstream &)

- Read(istrstream &)
- GetKeyField

GetInfo For each class, a file is used to store all its objects. We can use the name of the class as the file name. Each class should have a static object member of type Info. Info is a friend class to the Persistent Class. The constructor of class Info takes the file name and opens the database, the destructor closes the database. The method *GetInfo* gives out a reference to this static object. The Persistent Class uses this method.

Write The data of an object is packed in a string and stored under it's key. How to pack the data in a string is to be defined by each object in the method *Write(ostrstream &)*.

Read In the retrieval of an object, a string of bytes is retrieved and the data members of the object are to be constructed from the string of bytes. How to convert this data string into data members is to be defined by each object in the method *Read(istrstream &)*.

GetKeyField This method returns the key of the object. It return a pointer to a char string.

A.3 Packing data members of an object

In saving an object to disk, all data of the object is converted into a string of bytes and this string of bytes is stored on disk. While retrieving, a string of bytes is retrieved and converted to data members of the object.

In this section we give headers of some functions. These functions are used to write a data member of an object to a string of bytes and to read a data member from the string of bytes.

A.3.1 Data Members of Primitive Type

All the following functions return the number of bytes read(write) from(to) string stream.

- *WriteNumber*

The number to be written to a string can be of any type(int, unsigned int, short int, long int, float, double).

```
template<class Type>
int WriteNumber(ostrstream &os, Type number)
```

- *ReadNumber*

The number to be read from a string can be of any type(int, unsigned int, short int, long int, float, double).

```
template<class Type>
int ReadNumber(istrstream &is, Type &number)
```

- *WriteChar*

The char to be written to a string can be of any type(char, unsigned char).

```
template<class Type>
int WriteChar(ostrstream &os, Type character)
```

- *ReadChar*

The char to be read from a string can be of any type(char, unsigned char).

```
template<class Type>
int ReadChar(ostrstream &is, Type &character)
```

- *WriteString*

```
int WriteString(ostrstream &os, char *str)
```

- *ReadString*

Required memory for "str" is allocated inside the function.

```
int ReadString(istrstream &is, char *&str)
```

A.3.2 Data members of complex type

The data present in the data structures like array, linked lists etc,. can be converted into a string using the above write functions. The data string can be formatted into the data structure using the above read functions.

Eg:

/* This function converts a list (List<Type>) of node type (int, short int, long int, double, float, char) into data string */

```
template <class InfoType>
int WriteListOfNumbers(List<InfoType> *list,ostrstream & os)
{
   int totalBytes = 0;  // total bytes stored
   int noElements;
   if(list == NULL) {
     noElements = 0; // no elements in the list
     totalBytes += WriteNumber(os, noElements);
     return totalBytes;
```

```
    }
    noElements = list -> GetNoElements(); // number of elements in the list
    totalBytes += WriteNumber(os, noElements);
    Node<InfoType> *head = list -> GetHeader();
    while(head) {
        InfoType elm = head->GetInfo();
        totalBytes += WriteNumber(os, noElements);
        head = head -> GetNext();
    }
    return totalBytes;  // return total number of bytes saved
}
```

/* This function retrieves lists (List<Type>) of node Type (int, short int, long int, double, float, char) from the data string */

```
template <class InfoType>
int ReadListOfNumbers( List<InfoType> *&list, // retrieved list
                       istrstream & is
                     )
{
    int totalBytes = 0;
    int noElmnts = 0;
    totalBytes +=  ReadNumber(is,noElmnts);
    if(noElmnts == 0) list = NULL; // Zero elements in the list
    else list = new List<InfoType>; // create new list
    for(int i=0;i < noElmnts;i++) {
        InfoType ele;
        totalBytes +=  ReadNumber(is,elm);
        list -> Append(ele);   // appending to list
    }
    return totalBytes; // total bytes retrieved
}
```

A.4 An example persistent class

This example code of a Course class shows the supplementary code needed to support object persistence.

```
class Course : public Persistent {
public:
   Course() {}
   SetKeyField(char *cName);
   char *GetKeyField()  { return courseName; }
   Read(istrstream &);
   Write(ostrstream &);
   Info GetInfo() { return infoObjectl; }
private:
   char *courseName;
   int Units;
   static Info infoObject;
}

Info Course::infoObject("Course");  // all course object are stored in fi

Course::Write(ostrstream &os) {
      WriteNumber(os, Units);
      WriteString(os, courseName);
}

Course::Read(istrstream &is) {
 /* The data members should be read in the same order in which
  * they were written
  */
      ReadNumber(is, Units);
      ReadString(is, courseName);
}
```

The Course class inherits methods to Save, Retrieve and Remove an object from Persistent class.

To save an object, the method *Save* of the object is to be invoked.

Eg:

A trivial example

```
Course  c(''cs635'',4);
c.Save(); // the object  c  is stored
```

To retrieve an object: create the object, set key field, and invoke the method *Retrieve*. The method Retrieve returns a positive number if object with the specified object-id found on the disk, otherwise it returns zero.

Eg:

A example to show how to retrieve an object from disk.

```
Course *c;
c = new Course;
c->SetKeyField(''cs699''); // set object key
if( c->Retrieve() == 0) {      // retrieve from disk
   cout << " Object with given key not found ";
   delete c;
}
```

Appendix B

Template Library

This library gives headers of the template functions used in the generated code.

CreateLink

```
template<class leftType, class rightType>
void CreateLink(leftType, rightType)
```

This function creates a link between two objects. Pointers to objects to be linked are to be passed as arguments to this function. For more information please see section 3.1.2.

DeleteLink

```
template<class leftType, class rightType>
void DeleteLink(leftType, rightType)
```

This function deletes link between two objects. Object pointers are to be passed as arguments to this function.

GetPtrToObject

```
template<class Type>
void GetPtrToObject(char *ObjectId, Type &ObjectPtr)
```

This function gives a pointer to an object in *ObjectPtr* whose object key is *ObjectId.* If object with specified key is not present in the system, *ObjectPtr* will be NULL. When an object wants to send a message to another object whose object-id is known, it gets a pointer to the receiving object using this function.

GetPtrsToObjects

```
template<class Type>
void GetPtrsToObjects(List<Type> &ObjectPtr)
```

The function gives a list of pointers to objects of specific class type. This is used, when a message is to be sent to some objects of a particular type that satisfy a condition.

Eg

The following call to this function gives a list of pointers to all Course objects.

```
List<Course *>  *courseObjects;
void GetPtrsToObjects(courseObjects)
```

GetPtrToRelatedObjects

```
template<class Type>
void GetPtrToRelatedObjects(KeysObject &objectIds, List<Type> &objPtrs)
```

This function is used by an object to get pointers to its related objects of a particular type. Each object maintains object-ids of its related objects. For an example please see example 4 of section 3.4.

Appendix C

Notation in BNF

In this appendix a BNF grammar for the specification language is given.

```
systemspec      :       classlist genericmsgs processlist ;
classlist       :       classdecl
                |       classlist  classdecl
                ;
classdecl       :       CLASS  ':' classname  EOFLD option_list ;
option_list     :       /*nothing*/
                |       TYPE ':' ID EOFLD option_list
                |       GENERALISATION_OF ':' classname_list EOFLD option_list
                |       INHERITS ':' classname_list EOFLD option_list
                |       ATTR ':'  attr_list  EOFLD  option_list
                |       RELATED_WITH ':' rel_list EOFLD option_list
                |       AGGROF ':' agglist EOFLD option_list
                ;
classname       :       ID  ;
classname_list  :       ID
                |       classname_list ',' ID
                ;
```

```
attr_list       :       /* nothing */
                |        attr
                |        attr_list ',' attr
                ;
attr            :        attribute
                |        attribute '<' attrtype '>' /* <attr_type>*/
                ;
attribute       :        ID  ;
attrtype        :        ID
                |        c_list
                |        c_table
                ;
c_table         :        TABLE '[' attr_list ']'
;
c_list          :        '[' attrtype ']'
;
agglist         :        aggof
                |        agglist ','  aggof
                ;
aggof           :        ID '(' multiplicity ')' ;
rel_list        :        relation
                |        rel_list ',' relation
                ;
relation        :        ID '(' reltype  ',' multiplicity ')'
;
multiplicity    :        lmcity '-' umcity
                |        umcity
                ;
reltype         :        ID ;
```

```
umcity          :       'M'
                |       'N'
                |       CARDINALITY
                ;
lmcity          :       CARDINALITY ;
/*************************************************************************
                        GENERIC MESSAGES
/*************************************************************************/
genericmsgs     :       /* optional */
                |       gen_messages
                ;
gen_messages    :       GENERIC_MESSAGE SEP gmsglist ;
gmsglist        :       gmsg_decl
                |       gmsglist  gmsg_decl
                ;
gmsg_decl       :       gsender grecvr msg_id optlist ;
gsender         :       FROM ':' gen_clsname constraint EOFLD ;
grecvr          :       TO ':' gen_clsname constraint EOFLD ;
gen_clsname     :       '*'
                |       ID
                ;
/*************************************************************************
        ORDINARY PROCESS, GENERIC PROCESS, AND SUB PROCESS
/*************************************************************************/
processlist     :       gen_proclist sub_proclist proclist ;
sub_proclist    :       /* optional */
                |       sub_process
                |       sub_proclist sub_process
                ;
sub_process     :       SUB_PROCESS SEP pid EOFLD mesg_sequence ;
```

```
gen_proclist    :       gen_process
                |       gen_proclist gen_process
                |       /* optional */
                ;
gen_process     :       GENERIC_PROCESS SEP pid EOFLD
                        INPUTS ':' varlist EOFLD
                        mesg_sequence
                ;
varlist         :       ID
                |       varlist ',' ID
                ;
proclist        :       process
                |       proclist process
                ;
process         :       PROCESS SEP pid EOFLD mesg_sequence
                ;
pid             :       ID   ;
mesg_sequence   :       sequence
                |       mesg_sequence sequence
                ;
sequence        :       msglist  /* sequence */
                |       if_block
                |       loop_block
                ;
if_block        :       IF '(' conditionList ')' elsepart
                ;
elsepart        :       /* else is optional */
                |        ELSE '{' mesg_sequence '}'
                ;
loop_block      :       LOOP loopcond '{' mesg_sequence '}'
                ;
```

```
loopcond        :       field '(' listOrtable ')'
                |       '(' conditionList  ')'
                ;
field           :       ID ;
listOrtable     :       ID ;
msglist         :       msgdecl
                |       msglist  msgdecl
                ;
msgdecl         :       sender recvr msg_id  optlist ;
sender          :       FROM ':' classname constraint EOFLD ;
recvr           :       TO ':' classname constraint EOFLD ;
constraint      :      /* Constraint is Optional */
                |       '(' cond ')'
                ;
cond            :       lhs operator rhs
                |       reference
                ;
lhs             :       ID
                |       classname SEP ID
                ;
rhs             :       ID
                |       classname SEP ID
                ;
reference       :       ID ;
operator        :       EQUAL
                |       NOT_EQUAL
                |       LESSER
                |       GREATER
                |       LESSER_EQUAL
                |       GREATER_EQUAL
                ;
```

```
msg_id          :       MSG_ID ':' message  EOFLD ;
message         :       mesg_id  '(' ')'
                |       mesg_id '('  paramlist ')'
                ;
mesg_id         :       ID ;
paramlist       :       param
                |       paramlist ',' param
                ;
param           :       var
                |       var '<' type '>' /* < paramtype > */
                ;
var             :       ID  ;
type            :       ID
                |       m_list
                |       m_table
                ;
m_list          :       '[' type ']' ;
m_table         :       TABLE '[' paramlist ']' ;
optlist         :       /* nothng */
                |       msg_cond optlist
                |       msgtype optlist
                |       result optlist
                |       action optlist
                |       precondoptlist
                |       postcond optlist
                ;
msgtype         :       MSGTYPE ':' ID EOFLD  ;
msg_cond        :       MSG_COND  ':' mcond EOFLD ;
mcond           :       constraint   ;
precond         :       PRE_COND ':' conditionList EOFLD ;
```

```
conditionList   :       condition
                |       conditionList l_operator condition
                ;
l_operator      :       AND  /* && */
                |       OR   /* || */
                |       NOT  /* !  */
                ;
condition       :       RECVD mesg_id
                |       RECVD mesg_id OF proc_id
                |       RECVD RESULT_OF mesg_id
                |       RECVD RESULT_OF mesg_id  OF proc_id
                |       DONE_SUB_PROCESS mesg_id
                |       cond
                ;
/* We are ingnoring post conditions */
postcond        :       POST_COND ':' pcond EOFLD ;
pcond           :       ID ;

result          :       RESULT ':' res EOFLD ;
res             :       attr
                |       set_of_values
                ;
set_of_values   :       ID  '{' attr_list '}' ;
action          :       ACTION  ':' action_list EOFLD ;
action_list     :       actual_action
                |       action_list ',' actual_action
                ;
actual_action   :       DO_SUB_PROCESS sub_proc_id sub_proc_args
                |       DO_GEN_SUB_PROCESS sub_proc_id sub_proc_args gen_values
                |       RETREIVE  res
                |       SEND_MSG mesg_id to_recvr  OF proc_id
```

```
                |       SEND_MSG mesg_id to_recvr  /* mesg_id of current proc
                |       SEND_GEN_MSG mesg_id  to_recvr inputs
                |       SELECT_MENU_ITEM item
                ;
to_recvr        :       /* optional */
                |       ID ID              /* Eg: to CDGC */
                ;
gen_values      :       /* optional */
                |       '{' value_list '}'
                ;
value_list      :       value
                |       value_list ',' value
                ;
value           :       ID ':'  replacing_val ;
replacing_val   :       ID
                |       DOLLAR ID
                ;
inputs          :       '{' input_list '}' ;
input_list      :       var_value_pair
                |       input_list ',' var_value_pair
                ;
var_value_pair  :       TO    ':' ID
                |       TO_CONSTRAINT ':' constraint
sub_proc_args   :   /* no args for sub_process */
                |       '('  paramlist ')'
                ;
sub_proc_id     :       ID ;
proc_id         :       ID ;
item            :       ID ;
```

References

[Boo94] Grady Booch. *Object Oriented Design and Applications.* Benjamin Cummins, 1994.

[Bud91] Timothy Budd. *An Introduction to Object Oriented Programming.* Addison-Wesley, 1991.

[Lip] Stanley B. Lippman. *C++ Primer.* Addison-Wesley, 2 edition.

[Pap95] David M. Papurt. Automatic implementation of association. *Dr. Dobb's Journal,* Oct 1995.

[R+91] James Rumbaugh et al. *Object Oriented Modelling and Design.* Prentice Hall, 1991.

[Sar96] Burgula Ramanjuneya Sarma. A message based specification system for object oriented software construction. Master's thesis, IIT-Kanpur, Feb 1996.

[Shi88] Etsuya Shibayama. How to invent distributed implementation schemes of an object-based concurrent language. *OOPSLA 88 Proceedings*, Sep 1988.

[Str91] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley, 2 edition, 1991.

[Vij96] Maram Vijayanand. Translating message centric oo specification to rdbms. Master's thesis, IIT-Kanpur, March 1996.